UNIVERSIDADE FEDERAL DE ALAGOAS

Instituto de Computação
Ciência da Computação
Período 2018.1

Linguagem Sapphire

Especificação da linguagem para trabalho de compiladores sob orientação do professor Alcino Dall'Igna

Lucas Ribeiro Raggi
Wagner da Silva Fontes

Sumário

Forma Geral do Código Escrito em Sapphire

Todo arquivo fonte da linguagem deve ter a extensão “.sapp” para que possa ser reconhecido. Para a execução do programa, será necessário uma função principal (“main”), que será o ponto de partida.

func void main()**:**
    ...

end

Demais funções devem ser definidas acima desta.

Como mostrado acima, todos os escopos são iniciados a partir do marcador “:” e finalizados com a palavra reservada “**end**”. É altamente recomendado que o código seja corretamente identado para uma melhor legibilidade da linguagem, no entanto é opcional. Todos os comandos são finalizados com uma quebra de linha. E os comentários são somente por linha, e deve ser posicionado após o caractere “@”.

Características Léxicas

As variáveis e funções são nomeadas através de identificadores. O identificador só pode ser iniciado por letras, identificadores também não podem coincidir com palavras reservadas, caracteres especiais não são permitidos.

Os identificadores são case-sensitive, ou seja, nomes com letras maiúsculas e minúsculas tem diferença. O tamanho dos identificadores não podem superar 30 caracteres.

- Especificadores de tipo:
    - int
    - float
    - bool
    - char
    - str

- Função principal:
    - main

- Funções de entrada e saída:
    - input
    - show

- Retorno de função
    - return

- Comandos de condição ou de iteração
    - if
    - else
    - elif
    - while
    - for

- Operadores logicos:
    - and
    - or

Variáveis e Tipos

As variáveis devem estar declarada dentro de alguma função, ou seja, não há variáveis globais e são vinculadas estaticamente, portanto no momento de sua declaração deve ser especificado o tipo.

Tipos de variáveis:

- **float:** ponto flutuante representado em 32 bits.
- **int**: valor inteiro representado em 16 bits.
- **bool:** lógico, verdadeiro ou falso.
- **str:** string de caracteres.
- **char:** caractere.

Há ainda a possibilidade de criação de arrays para todos os tipos acima, no entanto cada array deve conter um único tipo de variável e sua declaração deve seguir o padrão:

tipo[tamanho] identificador

E o acesso a seus elementos:

… identificador[0]

As posições são marcadas a partir de zero.

Existe a função size() implementada pela linguagem que recebe um array de qualquer tipo e string e devolve o tamanho da mesma

Caso o tipo seja float a linguagem irá decidir se irá colocar mais casas a direita/esquerda ou ultilizara notação científica na representação.

Constantes Literais

Inteiro

Representados com 4 bytes. Podendo assumir valores dentro do intervalo −2.147.483.648 a 2.147.483.647.

Ponto flutuante

Ponto flutuante com precisão simples, representados em 4 bytes.
Ex.: 3.14159.

Booleano

Pode assumir somente dois valores: **true** ou **false**.

Caractere

Letras e símbolos de 8 bits (ASCII). Obrigatoriamente representados entre aspas simples.
Ex.: 'a', 'b', 'H', '^', '*','1','0'.

String

Cadeia de caracteres de 1 byte cada delimitado por aspas duplas.
Ex.: “All by myself...”.

Arranjos unidimensionais

Conjunto de elementos, todos do mesmo tipo, dentro de colchetes e separados por vírgula.
Ex.: [88,343,12], [“hello”, “oie”], [true, true, false].

Coerção

Para toda operação de concatenação de uma string com uma variável de outro tipo haverá coerção deste segundo tipo para string.

A outra coerção suportada pela linguagem é entre o tipo inteiro e ponto flutuante.

Quando um valor inteiro for atribuído a uma variável de ponto flutuante, esta terá o valor de sua parte decimal mantido em zero, e sua parte inteira idêntica à variável inteira atribuída. Já em caso de um valor em ponto flutuante ser atribuído a uma variável inteira, esta armazenará apenas a parte inteira do numeral, desprezando a parte decimal independente de seu valor.

Escopo

As variáveis são visíveis apenas na função em que foram criadas. Sapphire faz uso exclusivo de escopo estático.

Operadores

Aritmeticos:

+	Adição
-	Subtração
*	Multiplicação
/	Divisão
^	Exponenciação

Relacionais:

>	Maior que
<	Menor que
>=	Maior ou igual
<=	Menor ou igual

Relacionais (igualdade):

==	Igual
!=	Diferente

Logicos:

true	verdade
false	falso
not	negação

and	conjunção
or	disjunção

Concatenação de cadeia de caracteres:

&	Concatenação

Operações suportadas

int	atribuição, aritméticos, relacionais(ambos)
float	atribuição, aritméticos, relacionais(ambos)
bool	atribuição, lógicos, relacionais(igualdade)
char	atribuição, relacionais(ambos)
str	atribuição, relacionais(ambos), concatenação

Precedência e Associatividade:

A precedência é dada pela mais alta até a mais baixa na tabela a seguir as regras de associatividade é sempre da esquerda para direita, exceto para o operador de exponenciação. Neste caso particular será da direita para esquerda.

not
- (unário negativo)
^
*, /
+, -
<, <=, >, >=, ==, !=
and, or
=

Pode se utilizar parênteses para priorizar algum conjunto de operações, as operações dentro do parênteses sempre serão resolvidas primeiro

Funções

As funções são da seguinte forma:

```
func tipo_de_retorno nome_da_função (tipo parametro1, …):
      …
end
```

Toda função começa com a palavra reservada **func**, seguida do tipo de retorno, nome da função, parâmetros comandos e então a palavra reservada **end** que significa fim dos comandos daquela função.

O programa sempre começara a execução a partir da função com nome “main”, funções aninhadas não são permitidas, cada função deve ser declarada separadamente.

É obrigatório o uso da palavra reservada **return** para retornar o valor do tipo especificado em seu cabeçalho, a menos quando esse tipo de retorno é especificado como **void**. São permitidos como retorno qualquer tipo primitivo e arranjos unidimensionais.

As chamadas de função são do formato:

nome_da_função(parametro1, parametro2, …)

O modelo de passagem de parâmetro adotado para os tipos primitivos é de passagem de valor, já para arranjos unidimensionais é de passagem por referência.

Instruções

Estrutura condicional de uma e duas vias

if: é a estrutura de condição que permite avaliar uma expressão e, de acordo com seu resultado, executar uma determinada ação.

elif: é a estrutura caso seja necessário usar aninhamento de condições. Esta instrução será levada em consideração caso as condições anteriores à ela não forem satisfeitas.

else: caso nenhuma expressão condicional seja satisfeitas nas demais estruturas precedentes, os comandos presentes no escopo desta serão executados.

```
if ( false):
      ...
end
elif (false):
      ...
end
else:
      …
end
```

Estrutura iterativa com controle lógico

while: possibilita a execução de uma conjunto de instruções até que uma condição deixe de ser atendida.

```
while (true):
      …
end
```

Estrutura iterativa controlada por contador com passo igual a um caso omitido

for contador = valor, limite, passo:

…

end

for: esta estrutura tem seu funcionamento controlado a partir de uma lista de comando separados por vírgula. Onde o primeiro destina-se à atribuição do contador a um valor inteiro, o segundo é um inteiro que será o valor limite do contador, o "passo" pode ser omitido e então o contador será incrementado em 1.

Entrada

input: permite a leitura de dados para que sejam armazenados em variáveis. É possível ainda ler mais do que um único valor. Todos os caracteres encontrados entre aspas serão ignorados na leitura, devendo ser usado para fins de delimitação de cada valor os quais serão atribuídos às variáveis especificadas no comando. As variáveis devem estar separadas dos caracteres entre aspas por vírgula.

Apenas uma linha é capturada com esta instrução. Qualquer entrada fornecida pelo usuário dissimilar à especificada no comando acusará erro.

```
input(a , " " , string , "1" , b)
```

Sendo string do tipo str, a e b do tipo int.

Saida

show: exibe a variável passada por parâmetro, ou ainda uma mensagem marcada entre aspas duplas. Ou ainda concatenações destas. Dessa forma é permitida a saída de mais de uma variável na mesma instrução.

Em caso de impressão de mais de uma variável, é necessário separá-las pelo operador de concatenação. Já para somente uma variável, não importando seu tipo, é suficiente apenas o seu identificador.

Caso a variável seja ponto flutuante, serão destinadas duas casas decimais por padrão, e para alterar isto basta que seja especificada a quantidade de casas desejadas seguido de um "." e o identificador da variável.

```
show(string & " é igual a " & 4.c)
```

Sendo string do tipo str, e c do tipo ponto flutuante.

Exemplos

Hello world:

```
func void main():
    show(“Hello World”)
end
```

Fibonacci:

```
func void fib(int n):
    int a = 0
    int b = 1
    int aux
    int i
    for i = 0, i < n, i++:
        if i > 0:
            show(“,”)
        end
        if i == 1:
            show(“0”)
        end
        if i == 1:
            show(“1”)
        end
        else:
            aux = a + b
            show(aux)
            a = b
            b = a + b
        end
    end
end

func void main():
    int n
    input(n)
    fib(n)

end
```

Shell sort:

```
func void shellsort(int[] arr, int n):
    int i, j, t, temp
    for i = n/2, i > 0, i = i/2:
        for j = i, j < n, j++:
            temp = arr[j]
            for t = j, t >= i and (arr[t - i] > temp), t = t - i:
                arr[t] = arr[t - i]
            end
            arr[t] = temp
        end
    end
end

func void main():
    int size
    int[300] arr
    int i
    show("Digite o tamanho da sequência (limite de 300)")
    input(size)
    for i = 0, i < size, i++:
        input(arr[i])
    end
    shellsort(arr, size)
    for i = 0, i < size - 1, i++:
        show(arr[i] & ", ")
    end
    show(arr[size - 1])
end
```

Linguagem de Implementação

A linguagem para a implementação do analisador léxico e sintático escolhida pela equipe é JAVA.

Categorias dos Tokens

```
public enum TokenCategory{
        MAIN, ID, INT, FLOAT, BOOL, CHAR, STR, BEGIN_SCP, END_SCP, BEGIN_PARM,
        END_PARM, BEGIN_ARR, END_ARR, COMMENT, END_INS, SEPARATOR, CONST_INT,
        CONST_FLOAT, CONST_BOOL, CONST_STR, CONST_CHAR, INS_INPUT, INS_SHOW,
        INS_IF, INS_ELIF, INS_ELSE, INS_FOR, INS_WHILE, INS_RETURN, OP_ATRIB, OP_AND,
        OP_OR, OP_NEG, OP_ADD, OP_MULTI, OP_EXP, OP_RELAT, OP_REL_EQ, OP_CONCAT;
}
```

Tokens: categorias e expressões regulares

Lexema	Expressões
MAIN	‘main’
ID	‘[a-zA-Z][a-zA-Z0-9]{1-30}’
INT	‘int’
FLOAT	‘float’
BOOL	‘bool’
CHAR	‘char’
STR	‘str’
BEGIN_SCP	‘:’
END_SCP	‘end’
BEGIN_PARM	‘(‘
END_PARM	‘)’
BEGIN_ARR	‘[‘
END_ARR	‘]’

COMMENT	‘@’
END_INS	‘\n’
SEPARATOR	‘,’
CONST_INT	‘[:digit:]+’
CONST_FLOAT	‘{consNumInt}+\.{consNumInt}+’
CONST_BOOL	‘ true | false ’
CONST_CHAR	‘\’[[:alpha:][:digit:][:graph:][:space:]]\’’
CONST_STR	‘”[[:alpha:][:digit:][:graph:][:space:]]*”’
INS_INPUT	‘input’
INS_SHOW	‘show’
INS_IF	‘if’
INS_ELIF	‘elif’
INS_ELSE	‘else’
INS_FOR	‘for’
INS_WHILE	‘while’
INS_RETURN	‘return’
OP_ATRIB	‘=’
OP_AND	‘and’
OP_OR	‘or’
OP_NEG	‘not’
OP_ADD	‘+ | -’
OP_MULTI	‘* | /’
OP_EXP	‘^’
OP_RELAT	‘< | > | <= | >= ’
OP_REL_EQ	‘ == | != ‘
OP_CONCAT	‘&’